Handwritten digits recognition using OpenCV

Machine Learning in Computer Vision (CS9840), final project

Vincent Neiger, Western University

January 28th, 2015

Abstract

The automated recognition of handwritten digits is a largely studied problem which connects the fields of Computer Vision and Machine Learning and has many applications in real life. In this project, I detail an introductory investigation of the performance of classification in several contexts. Namely, relying on the OpenCV implementations of k-Nearest Neighbor, Random Forests, and Support Vector Machines classifiers, I compare the error rates obtained using several preprocessings of the dataset as well as various choices of features. The best obtained error rate on the MNIST dataset [2] is 0.81%, obtained with a Support Vector Machines classifier with a Gaussian kernel, using deskewing and blurring as preprocessing and using as features a combination of pixel intensities and histograms of orientations of gradients.

1 Introduction

Context. After some training, humans are able to communicate through handwriting, and reading is something they can do very efficiently even when the text has been written with a poor handwriting. The automation of recognizing the characters in handwritten texts obviously has many applications, so that this practical problem has been studied about as soon as we had enough technology to consider it feasible. Even when restricted to the easier case of recognizing handwritten numbers, this obviously has many real-life applications, some of which are already effective: for example, the automated recognition of postal codes, the automated processing of some administrative forms, and the automatic recognition of amounts on bank cheques.

Problem. In this kind of automation, the first step is to isolate the digits, which is usually made easier by providing strict boundaries which should contain them (Figure 1). In this context, the main problem which remains to be solved is the one we will focus on in the rest of this report: the *recognition of isolated handwritten digits*. More precisely, the problem is the following: write some program which, when given a clear image containing a single digit in some fixed format, predicts which digit is in the image. Two properties will make such a program a satisfactory one: its accuracy in predictions and its running time.

Approach and goals. The best existing classifiers use tools from Machine Learning [1, 2] to train the classifier so that it has a good accuracy on new unknown input samples; this involves tools from Computer Vision to compute features associated to an image. I followed this approach, trying to replicate some of the results in previous works cited on the webpage [2]. My goals were to learn how to use some programming tools related to Machine Learning and Computer Vision, and to

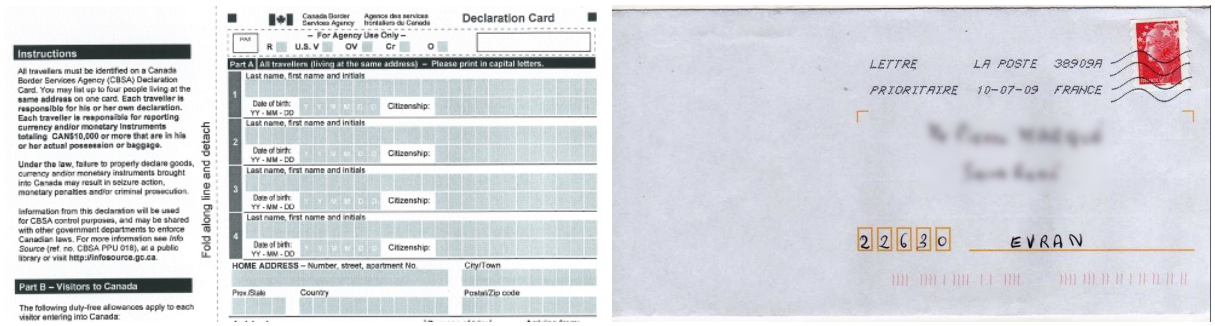

Figure 1: Automated digit recognition. *On the left:* administrative form (Canadian immigration declaration card). *On the right:* postal code 22630 on envelope (French postal services).

use them in order to experiment the influence of the choice of classifier, of features and of image preprocessings on both the global error rate and the timings of the classifier.

Outline. First, we briefly present the OpenCV library in Section 2, and we give details about the dataset and the classifiers used in this project. Then, in Section 3 we describe the different types of feature vectors that are studied in the experiments. Finally, in Section 4 we explain how some preprocessing of the images and some artificial enlargement of the training set led to improvements of the classification for some classifiers.

2 The OpenCV library, dataset and classifiers

Implementation details. Looking for libraries which would help me on one hand to process images and compute features, and on the other hand to run efficient implementations of some well-known classifiers such as k-Nearest Neighbor or Support Vector Machines, I ended up using *OpenCV* [3], the Open Computer Vision library. This open-source, cross-platform library for C, C++, Python, and Java, focuses primarily on Computer Vision algorithms, but it also has a Machine Learning module which includes the classifiers mentioned above as well as some others.

I chose to work with the C++ interface of OpenCV, and I found this library quite easy to learn thanks to the complete and clear documentation and the many examples provided on the official website or elsewhere on the web. All timings in the rest of this report were done using the attached code and `g++-4.8` compiler, on an HP Z600 desktop computer with 4 CPU Intel Xeon E5620 at 2.40GHz, running a Linux Ubuntu 14.04 64 bits operating system.

Dataset. A commonly used dataset for handwritten digit recognition named MNIST can be found on Y. Lecun website [2]. It is a collection of 70,000 digits written by 750 different people among Census Bureau employees and high school students. Digits were size-normalized, centered by center of mass, and stored sequentially as grayscale 28×28 images in a binary file. These digits were split into two disjoint sets: a training set with 60,000 images and a testing set with 10,000 images. The resulting datasets are provided at [2] along with labels (a single digit for each image); Figure 2 shows the first 100 digits in the training set.

This ready-to-use dataset is the data I used in all the experiments reported below. In my code, the data is processed in the following order:

Figure 2: MNIST dataset: first 100 digits in the training set

- a *parser* extracts the images and labels from the binary files,
- in case of training data, the training set may be *augmented with artificial training samples* obtained by rotating, scaling and shifting the images,
- preprocessing such as deskewing and blurring may be applied to the images,
- the chosen type of feature vectors are computed,
- feature vectors and the corresponding labels are stored.

Flexibility is offered to the user by providing a structure called `FeatureParams`, which can be used to specify whether and how to add artificial training data, the wanted type(s) of preprocessing, the type and size of features used, etc... This processed data can eventually be passed to some classifier, which can either train on these labelled samples, or classify them and check accuracy thanks to the labels.

Classifiers. In my experiments, I focused mainly on four classifiers.

- k-NN: a **k-Nearest Neighbor** classifier, using $k = 3$ since experiments showed me this choice almost always yielded a better accuracy than the other values I tried for k. As we will see in Sections 3 and 4, this classifier gives very good error rates; however, when it comes to the time performance for classifying, it is by nature quite slower than the three other classifiers below. Along with the fact that it stores all training samples, this might limit the practical use of this classifier.
- RF: a **Random Forest**, that is, a collection of random trees classifiers using majority vote from the output of the trees, which are trained on randomly chosen subsets of the training set. Many parameters can be tuned, such as the maximum depth of the trees, the maximum number of trees, and concerning learning: the termination criteria, how many samples at least must represent a node before it can be split, etc. The parameters I used are inspired from [4] and my own experiments; compared to the default parameters in OpenCV, this choice improved the error rate and reduced the running time, more or less significantly depending

on preprocessing and feature types. For instance, in the specific case of Raw features and deskewing (explained in later sections), the default parameters required 3m50s of computation with an error rate of 15.02%, while using better parameters the computation time was reduced to 2m33s and the error rate to 6.15%. Whatever parameters I tried, I never reached error rates below 3% using RF; however they are remarkable for their excellent time performance for classifying.

- SVM-RBF: a **Support Vector Machine** using a **Gaussian Radial Basis Function kernel**. To choose parameters for this classifier, I used the `train_auto` function of OpenCV, which splits the training set into a certain number of subsets, and perform cross-validation to optimize the parameters, by repeated training and testing using each time one of the subsets as testing set and the others as training set. This requires a lot of computations, so that I ran this cross-validation only on smaller randomly chosen subsets of the training set (from 1,000 to 10,000 samples). This led me to choose parameters $C = 10$ and $\gamma = 0.01$. SVM-RBF gave, in my opinion, the best results among the four classifiers: it allows to reach excellent error rates (below 1%) while requiring a quite shorter running time for classifying than k-NN.
- SVM-POLY: a **Support Vector Machine** using a **Polynomial kernel**. For this classifier, I fixed the polynomial degree to 5, following previous work [1, 2]; I then used `train_auto` to auto-tune other parameters, similarly to the SVM-RBF case. This led me to use $C = 0.1$ and constant shift in kernel polynomial $c_0 = 19.6$. Using SVM-POLY, I achieved good error rates with good classification running time.

3 Raw features and HOG features

In this section, the different kinds of feature vectors used are presented, and first experimental results are given. These error rates and running times should be put in perspective by comparing them to those in Section 4.

Raw features. At first, I considered the most straightforward feature vector: the raw 784 intensity values of the 28×28 grayscale image. If I had not looked at previous works results, I wouldn't have expected any acceptable classification using this very basic choice of feature vectors; this intuition was completely wrong. Indeed, from the table in [2], it seems reasonable to expect an error rate below 5% using k-NN or SVM-RBF.

And indeed, using k-NN, the error rate was 2.95%, which means that only 295 digits were misclassified out of 10,000 in the testing set. The experimental results concerning raw features are presented in Table 1; all running times are in seconds. The fifth column indicates the time used to compute features for the testing set, which is very fast in the case of Raw features. In Table 1, we see that the best error rate is 1.77%, obtained using SVM-RBF. Besides, RF has an excellent classification time.

It should be noted that when using SVM classifiers — especially with RBF kernel, but also to a lesser extent with a polynomial kernel — I had to normalize the feature vectors. The reason is that using raw values from 0 to 255 induced underflow which strongly biased the results (the obtained error rate of SVM-RBF on Raw features without normalizing was 89.9%, that is, not better than a classifier which would output a random digit for every testing sample). In what follows, all experimental results concerning the SVM-RBF classifier involve a normalization of each feature vector, so that its minimum component is 0 and its maximum component is 1, using only a

constant shift and a constant scaling; normalization with respect to norms L1 and L2 respectively did not give as good accuracy as this one.

Classifier	Error rate	Time (train)	Time (classify)	Time (features)
k-NN	2.95%	0.085	161	0.028
RF	6.15%	136	0.094	0.026
SVM-RBF	1.77%	307	27	0.027
SVM-POLY	11.4%	62	6.8	0.027

Table 1: Experimental results using Raw features.

HOG features. The second type of feature I focused on is called HOG [6], short for *histograms of oriented gradients*. Two parameters are used to define properly the HOG of an image: f the number of folds, and b the number of bins in the histograms. To build the feature vector, we first compute all horizontal and vertical gradients of the image and deduce their orientation (discarding the magnitude), then we use a $f \times f$ grid to define f^2 blocks composing the image, and finally for each block we compute b-bins histograms of the orientation of the gradients in this block. Here, for simplicity f is always a divisor of 28 (the width and height of the images) and the f^2 blocks are equal-sized. The feature vector consists of the concatenation of these histograms and thus contains $f^2 \cdot b$ features.

Figure 3 gives an illustrated example using $f = 2$ and $b = 22$ (note that the histogram on the right does not correspond to the gradients on the digit image on the left).

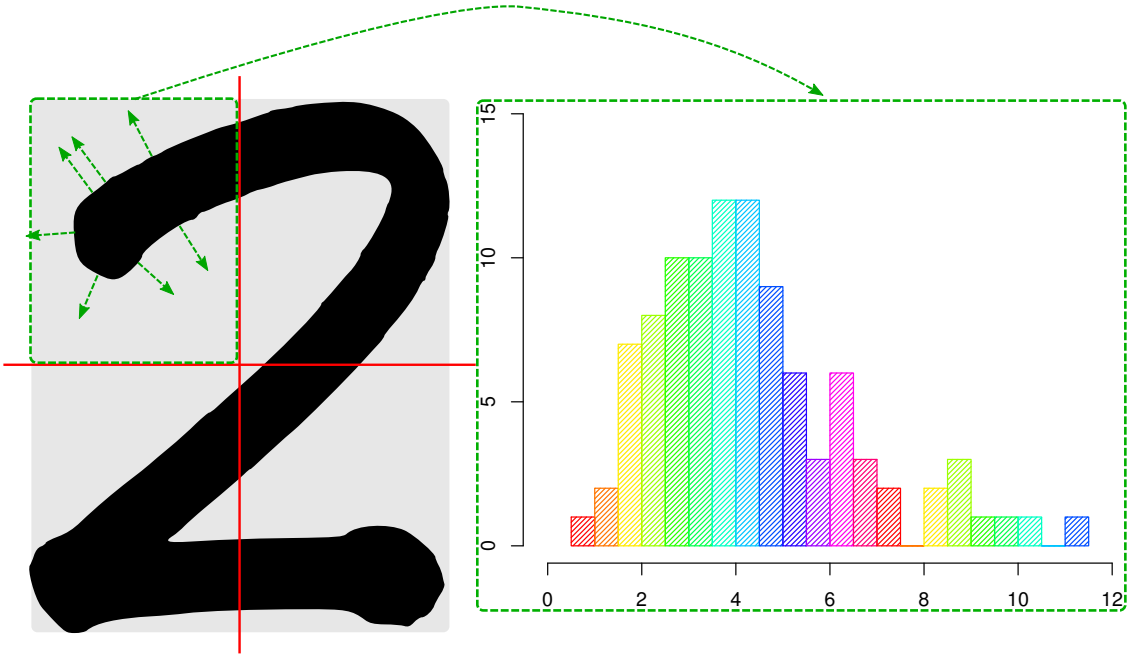

Figure 3: HOG features of digit 2, with 2-folding and 22 bins. This computation is repeated for each of the four blocks and the four obtained histograms form the feature vector.

Some experimentation led to the conclusion that in most cases, the best accuracy is obtained

with $b = 12$ bins; all error rates given thereafter concerning HOG features were computed with $b = 12$, and three sets of parameters are used:

- **HOG2**, with $f = 2$ and $b = 12$, thus 48 features per image,
- **HOG4**, with $f = 2$ and $b = 12$, thus 192 features per image,
- **HOG7**, with $f = 2$ and $b = 12$, thus 588 features per image.

The results are presented in Table 2 below. Unsurprisingly, the error rates for HOG4 are better than those for HOG2, and the error rates for HOG7 are better than those for HOG4; we have similar conclusions concerning running times because of the different feature vector sizes between HOG2, HOG4 and HOG7. What is more interesting is that HOG4 is competitive against Raw features in terms of error rates but requires to perform a considerably smaller amount of computations, given the quite smaller feature vectors. Besides, for every classifier using HOG7 leads to better error rates than Raw features, still using slightly fewer features; the time needed for training SVM-RBF decreases by a factor 2, as well as the time for classifying. This experiment illustrates the importance of choosing good features, for efficiency of both computation and classification.

Features	Classifier	Error rate	Time (train)	Time (classify)	Time (features)
HOG2	k-NN	5.91%	0.049	14	0.42
HOG4	k-NN	3.01%	0.27	46	0.73
HOG7	k-NN	2.30%	0.82	137	1.32
HOG2	RF	5.88%	15	0.090	0.43
HOG4	RF	4.56%	38	0.064	0.74
HOG7	RF	4.61%	129	0.093	1.32
HOG2	SVM-RBF	6.77%	92	5.3	0.49
HOG4	SVM-RBF	2.69%	131	8.6	0.78
HOG7	SVM-RBF	1.49%	151	14.0	1.34
HOG2	SVM-POLY	46.83%	2.92	0.21	0.43
HOG4	SVM-POLY	18.93%	16.2	1.23	0.73
HOG7	SVM-POLY	6.67%	43.4	4.33	1.28

Table 2: Experimental results using HOG features.

Other features. Another type of feature vector I used is the simple concatenation of Raw features and HOG features; I called them *RawHog* features. This allowed me to get better error rates without increasing too much the length of the feature vector, as detailed in Section 4.

Before focusing on HOG features, I had considered the more straightforward feature vector containing the orientation of the gradient for every pixel in the image. This gives a vector of 784 features; using these features I reached error rates such as 3.90% with k-NN and 1.85% with SVM-RBF (both with deskewing, blurring and normalizing, explained below). However, these results using orientations of gradients were not as good as for example those with HOG4, although the latter uses a quite smaller number of features (192 versus 784), this is why they will not be considered in the rest of this report. To observe the obtained error rates and timings, the reader can refer to the file `results_og_deskew_blurring.dat` in the attached project code folder.

In another direction, I tried using SIFT (Scale-invariant feature transform), without much success since the error rates were quite worse than with Raw or HOG4 features. However, this is

very likely due to insufficient understanding on my behalf about SIFT itself, and consequently about how to transform OpenCV's SIFT algorithm output into good feature vectors. This is certainly not due to an inherent limitation of SIFT itself since there are some results in the literature [5] announcing error rates as low as 1.5% using SIFT and SVM-RBF.

4 Improving the error rate using preprocessing, combining features, and creating artificial training data

Preprocessing: deskewing the images. This preprocessing was inspired from [1], where the authors write

> In the second version of the database, the character images were *deskewed* and cropped down to 20×20 pixels images. The *deskewing* computes the second moments of inertia of the pixels (counting a foreground pixel as one and a background pixel as zero) and shears the image by horizontally shifting the lines so that the principal axis is vertical.

In my code, I followed this approach, using OpenCV functions to compute moments and applying the deduced rotation to the image. However, in my code the images are not cropped, and it should be noted that this could be tried to provide further improvement, with a slightly faster and more accurate classification. The effect of deskewing on the images can be observed on Figure 4.

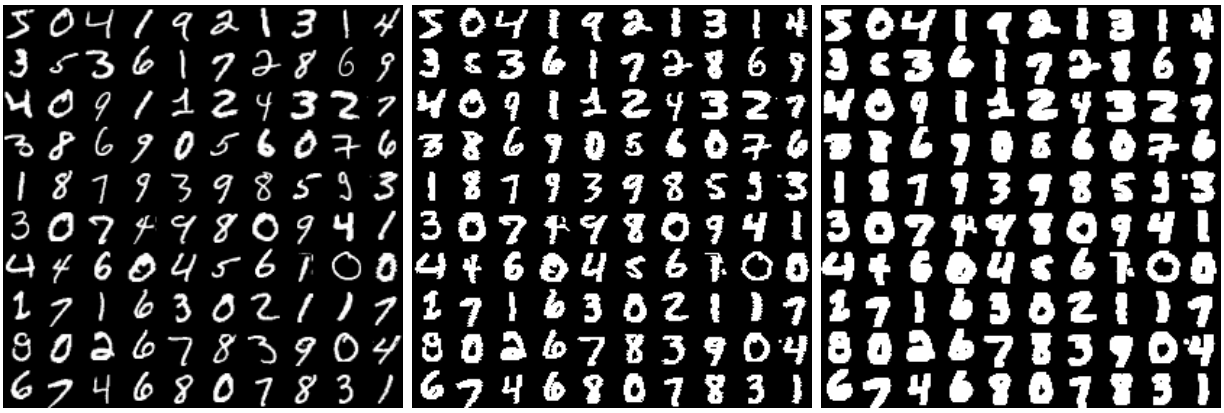

Figure 4: First 100 digits in the MNIST training dataset. *Left:* before deskewing. *Center:* after deskewing, without blurring. *Right:* after deskewing and blurring.

The idea behind this preprocessing is that different people's writings are more or less slanted, but rotating the digits from a slanted writing so as to deslant them gives digits that are very similar to digits from a non-slanted writing. This appears very clearly when looking at the digits "1" in Figure 4. Thus, deskewing provides a way to increase similarity between two samples representing the same digit and thus may simplify and improve the training stage. Naturally, for this to work, the exact same deskewing has to be applied on the images of the training set and on the images of the testing set. This preprocessing yielded better classification, as we can observe in Table 3.

Nevertheless, as we can notice on the center image of Figure 4, deskewing can increase the noise at edges of the digits. The intuition is that this may influence the classification, at least for features

based on the orientations of gradients (although using histograms already provides a way to smooth out this noise). To reduce this side-effect of deskewing, I applied a Gaussian blurring filter with kernel size 3 after deskewing. Error rates using deskewing and blurring are given in Table 3. To observe the improvements caused by this preprocessing, they should be compared to error rates in Tables 1 and 2. Table 3 excludes the RF classifier, for which deskewing led to very similar error rates as without preprocessing. It is worth noting the very poor error rates for SVM-POLY, which I have not investigated further; they may be caused by bad choices of parameters (and then could be improved using cross-validation on the training data as hinted at in Section 3), or by overfitting the training data, or both. Concerning the running times, they are very close to those in Tables 1 and 2, with the single change that there is additional time for preprocessing the images; deskewing and blurring the 10,000 images of the testing set takes only about 0.5s.

Features	Classifier	Deskewing	Deskewing & Blurring
Raw	k-NN	1.88%	1.72%
HOG4	k-NN	2.41%	2.00%
HOG7	k-NN	1.83%	1.59%
Raw	SVM-RBF	1.23%	1.18%
HOG4	SVM-RBF	2.02%	1.60%
HOG7	SVM-RBF	0.99%	0.95%
Raw	SVM-POLY	10.01%	31.28%
HOG4	SVM-POLY	10.32%	53.34%
HOG7	SVM-POLY	3.30%	3.48%

Table 3: Error rates using preprocessing.

Combining features. Trying other ideas to improve the classification, I used another type of features which combines Raw and HOG features. In this experiment, the feature vector is simply the concatenation of Raw features and HOG features, which gives the following characteristics (using notation from Section 3):

- **RawHog2**, with $f = 2$ and $b = 12$, thus $784 + 48 = 832$ features per image,
- **RawHog4**, with $f = 2$ and $b = 12$, thus $784 + 192 = 976$ features per image,
- **RawHog7**, with $f = 2$ and $b = 12$, thus $784 + 588 = 1372$ features per image.

Using these features, I only experimented with the best classifiers until now: SVM-RBF and k-NN, without changing the parameters used for SVM-RBF despites the new situation. This led to some improvement with in particular the best error rate over all my experiments: 0.81% using SVM-RBF with deskewing and blurring, with RawHog7 features. The results are presented in Table 4.

Features	Classifier	Deskewing	Deskewing & Blurring
RawHog4	k-NN	2.06%	1.69%
RawHog7	k-NN	1.84%	1.41%
RawHog4	SVM-RBF	1.05%	0.99%
RawHog7	SVM-RBF	0.93%	0.81%

Table 4: Error rates using preprocessing and combined features.

Augmenting the training set with artificial samples: distorted images. Finally, another intuitive way to reduce the error rate is to use more training data; once more inspired by [1], I tried to augment the training dataset by adding artificial samples to it. Here, an artificial sample is a real sample to which we have applied a random affine transformation which is a combination of rotation, scaling, and shifting. After some attempts, the following parameters were chosen for all experiments:

- rotation and scaling are applied before shifting,
- rotation with angle in $[-\pi/9, \pi/9]$ and with center the center of the image,
- scaling factor in $[0.9, 1.2]$,
- horizontal and vertical shifting of at most 2 pixels in each direction.

My choice was to give only a small role to shifting because the digits in the images of the MNIST dataset are centered.

In the first set of experiments, for each sample in the training set, I added either 9 rotated-scaled samples (angle in $\{-\pi/9, 0, \pi/9\}$ and scaling factor in $\{0.9, 1, 1.2\}$); or 9 shifted samples (horizontal shift in $\{-2, 0, 2\}$ and vertical shift in $\{-2, 0, 2\}$); or combining these, 18 transformed samples. In all cases, the image originally in the dataset is always one of these samples, with angle 0, scaling 1 and shifts 0.

Then, I ran the classifier exactly as before but with this augmented dataset. While this did significantly improve the error rates when using small subsets of the training set — sometimes dividing the error rate by 2 or 3 —, the benefit is quite reduced when using the whole training set. Table 5 below shows the error rates obtained in the rotation-scaling experiment, which had the best error rates among the three. k-NN and RF show some improvements thanks to this enlarged dataset, while SVM-RBF performs better with the real dataset.

Besides this slight improvement in error rate for some classifiers, the running times are dramatically impacted by using a 9 times larger training set. For example, using Raw features: the training time for k-NN is about 1 second, but the classifying time about 40 minutes; the classifying time for RF is about 0.3s but the training time about 35 minutes; and concerning SVM-RBF, the classifying time is around 2 minutes while the training time exceeds 3 hours, with significant memory usage. Surprisingly, SVM-RBF on RawHog7 features was still feasible, with barely longer running times and with a good error rate.

Features	k-NN	SVM-RBF	RF
Raw	1.57%	4.59%	4.15
HOG4	1.79%	2.01%	3.08%
HOG7	1.22%	2.57%	3.18%
RawHog7	1.49%	2.52%	3.51%
RawHog7	1.24%	0.92%	3.60%

Table 5: Error rates using 9-rotations-scalings-enlarged training set.

In the second set of experiments, for each real training sample, the training set is augmented with a fixed number c of randomly distorted samples, by choosing a random rotation satisfying the constraints above (the angle is a random floating-point number in $[-\pi/9, \pi/9]$), and similarly a random scaling and a random shifting. The conclusions were similar to the first case: a modest

improvement is observed for k-NN and RF but not for SVM-RBF, while the running times are seriously increased.

Finally, I tried two solutions based on majority vote; which correspond to the source files whose names start with `multi`. First, using the method of the previous paragraph, we get a training set of size $60000(c+1)$. Then, this set is split into $c+1$ randomly chosen disjoint subsets of size 60,000; training and classifying are performed on each subset and the final decision is taken by majority vote. The results were slightly disappointing, with the same conclusions as in other attempts above except that SVM-RBF performed a little better.

Second, elaborating on this method, six of the best classifiers we have encountered in this report are considered, namely SVM-RBF and k-NN on Raw, HOG7, and RawHog7 features. Then, using the method of the previous paragraph to augment the training set and split it into $c+1$ subsets of size 60,000, their results give $6(c+1)$ possible classifications for each sample to classify ($c+1$ for each of the six classifiers). Then, the final decision is taken by majority vote on those classifications, the voting being slightly influenced by weights which depend on the assumed accuracy of the classifier. Once again, despites good results, this did not provide major improvement over previous classifiers; furthermore, the best error rate was not better than 0.81% and was obtained with $c = 0$, i.e., using no added artificial data. Experimental results, with $c+1 \in \{1, 2, 3, 4\}$, were computed using the function `bench_multi` in the file `main.C` and can be found in files whose names start with `multi_fold`; a summary is presented in Table 6.

$c+1$	1	2	3	4
Number of runs	12	6	4	3
Average error rate	0.84%	0.90%	0.91%	0.93%
Average time per run	50mn	2h	3h	3h45

Table 6: Error rates using augmented training set and majority vote on best classifiers.

5 Conclusion

In this project, I have had the opportunity to study some concepts of Machine Learning in Computer Vision from a fruitful practical point of view. The influence of the classifier, of its parameters, of well-chosen data preprocessing, and of a good choice of features, on running times and accuracy clearly appeared in the numerous experiments I made. The experiments about preprocessing — deskewing and blurring — gave excellent accuracy with quite good running times, far beyond my initial expectations. However, even though the experiments about augmenting the training set with artificial samples showed that Random Forests could reach a 3% error rate, they did not give any further improvement concerning the best obtained accuracy. The intuition in trying these majority vote classifiers came from the remarkable fact that only about half of the 81 samples that are misclassified by the SVM-RBF classifier are also misclassified by the k-NN classifier. Still, while this shows some room for a minor improvement of the error rate 0.81%, it seems now clear in my opinion that more elaborate Machine Learning tools and ideas are required to get to the best error rates in the literature, below 0.3%.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE, 1998, Volume 86, Issue 11, pp. 2278–2324.

[2] http://yann.lecun.com/exdb/mnist/ The MNIST database webpage on Y. Lecun website.

[3] http://opencv.org/ The OpenCV library website.

[4] S. Bernard, L. Heutte, and S. Adam, *Using Random Forests for Handwritten Digit Recognition*, Proceedings of ICDAR 2007.

[5] K. Kavukcuoglu, M. Ranzato, R. Fergus, and Y. LeCun, *Learning Invariant Features through Topographic Filter Maps*, Proceedings of CPVR 2009.

[6] N. Dalal, and B. Triggs, *Histograms of oriented gradients for human detection*, Proceedings of CPVR 2005.